Table of Contents

OBSOLETE.....1

Introduction.....2
- Concept.....2
- Layout.....2

Commissioning.....4
- Testing with old COOL.....4

Development workflow.....5
- Using overlays.....5
- Add files to a GitCondDB.....5

Operations.....7
- Cloning new global tags from COOL/SQLite.....7

Troubleshooting.....8
- What to do when cherry-picking fails.....8

References.....9

OBSOLETE

This page has been superseded by: http://lhcb-core-doc.web.cern.ch/lhcb-core-doc/GitCondDB.html

Introduction

GitCondDB is the new backend for condition data in LHCb.

Concept

The idea is simple: map time evolution of conditions data on a filesystem hierarchy and use Git internal database to manage versions and tags.

Layout

The conditions database uses a 3-dimensional structure, where the axis are:

- condition id (usually, path to an XML file)
- version (tag, commit id, branch name, ...)
- Interval Of Validity (IOV, range of event times a value should be used for)

The first two axis, id and version, map directly to the two axis of a Git repository (path and version), while for the IOVs we use a special convention:

- if a condition id points to a file, there is only one value for the whole span of all event times (from 0 to MAX)
- if a condition id points to a directory, it must contain a file called `IOVs` in which each line is constituted by the start of the IOV for a value and the relative path to the file containing the value, or payload (the end of the IOV is defined by the next line or set to MAX for the last line of the file. If the path to the value points to a directory, the algorithm to extract the value is repeated on the contained `IOVs` file and the deduced IOV is truncated to the boundaries deduced in the previous iteration.

To explain better how the IOV definitions work, let's assume we need to find the value for condition `Conditions/MyDetector/Cond.xml` at event time 1234

- if `Conditions/MyDetector/Cond.xml` is a file: the value is the content of the file and the IOV is [0, MAX)
- if `Conditions/MyDetector/Cond.xml` is a directory (simple case)
    1. we open `Conditions/MyDetector/Cond.xml/IOVs` where we find something like

0	value1
100	value2
200	value1
1000	value3
2000	value4

1.
    1. the value is the content of `Conditions/MyDetector/Cond.xml/value3` and the IOV is [1000, 2000)

- if `Conditions/MyDetector/Cond.xml` is a directory (nested case)
    1. we open `Conditions/MyDetector/Cond.xml/IOVs` where we find something like

0	value1
100	subdir1
1200	subdir2

1.
    1.
        1. we open `Conditions/MyDetector/Cond.xml/subdir2/IOVs` to look for IOVs in [1200, MAX), where we find something like

1000	../value3
2000	../value4

1.
    1.
        1. the value is the content of `Conditions/MyDetector/Cond.xml/value3` and the IOV is [1000, 2000)
    2. the value is the content of `Conditions/MyDetector/Cond.xml/value3` and the IOV is [1200, 2000) (i.e. the IOV found in the subdir, bounded to the IOV of the subdir itself)

Commissioning

The old COOL-based CondDB partitions have been cloned (only global tags) to Git repositories at https://gitlab.cern.ch/lhcb-conddb, with a regularly updated mirror on CVMFS (`/cvmfs/lhcb.cern.ch/lib/lhcb/git-conddb`).

The ONLINE partition is regularly updated with conditions produced at the Pit (PVSS and automated alignment and calibration).

Testing with old COOL

Any software project based on LHCb v42r2 or later uses GitCondDB by default, but the old COOL CondDB can be used just setting one environment variable:

```
export NO_GIT_CONDDB=1
```

Development workflow

Developing changes to GitCondDB is not different from a standard Git-based development workflow:

```
export GITCONDDBPATH=$(pwd)
git clone ssh://git@gitlab.cern.ch:7999/lhcb-conddb/DDDB.git
cd DDDB
git checkout -b my-changes
# edit
git commit -a -m 'super improvement'
git push -u origin my-changes
# create a merge request
```

To review you changes before committing them, you can use the CondDBBrowser:

```
lb-run LHCb/latest CondDBBrowser $GITCONDDBPATH/DDDB
```

The environment variable `GITCONDDBPATH` is enough to tell an LHCb application to use the GitCondDB repositories from that directory, then you need to just tell it to use the right version, adding the following lines to an option file:

```
from Configurables import CondDB
CondDB ().Tags['DDDB'] = 'my-changes'
```

You can use as tag anything Git understand as a commit identifier (tag, branch name, commit it...) plus the empty string, which instructs the application to use the checked out files rather than a commit.

⚠ tags and branches for the Upgrade geometry **must** be prefixed with `upgrade/` and you should use the branch *upgrade/master* instead of *master*.

Using overlays

When working with the checkout of whole databases is unpractical (e.g. ONLINE), it's possible to override individual files or directories in the database from a local directory, let's call it an *overlay*.

An overlay has to be a Git repository, even if it's only a directory in which `git init` was called, then it must be declared in the configuration through the "addLayer()" method of the CondDB configurable:

```
from Configurables import CondDB
CondDB ().addLayer('/path/to/my/overlay/dir')
```

Add files to a GitCondDB

For each IOV to add:

- create a directory with the files to be added for one IOV, with the appropriate hierarchy
- use the script `add_files_to_gitconddb.py` (from LHCb) to copy the files

For example, I want to add a new version of `MomentumScale.xml` in LHCBCOND at `Conditions/LHCb/Calibration/MomentumScale.xml` starting from 2018-11-20 00:00:00 local time, this is how to proceed:

```
mkdir -p tmp1/Conditions/LHCb/Calibration/
emacs tmp1/Conditions/LHCb/Calibration/MomentumScale.xml
# beware, the clone needs about 2 GB of disk space (at the end of 2018)
git clone ssh://git@gitlab.cern.ch:7999/lhcb-conddb/LHCBCOND.git
```

```
cd LHCBCOND
git checkout --no-track -b new-mom-scale origin/dt-2018
lb-run -c best LHCb/v50r1 add_files_to_gitconddb.py --debug --since $(date +%s000000000 -d 2018-1
git add .
git commit -m 'New MomentumScale'
git push -u origin new-mom-scale
```

Operations

Cloning new global tags from COOL/SQLite

See Porting changes from COOL to GitCondDB

Troubleshooting

What to do when cherry-picking fails

From time to time, when one tries to propagate a merge request to other branches using gitlab web interface, the red error message will show up "Sorry, we cannot cherry-pick this merge request automatically.
This merge request may already have been cherry-picked, or a more recent commit may have updated some of its content.", and the cherry-picking just cannot proceed as intended. In this case, please use the following command lines to fix things manually:

```
# Clone the git repository:
$ git clone https://:@gitlab.cern.ch:8443/lhcb-conddb/SIMCOND.git

# look for the merge commit in the original branch and locate the correct commit reference, Sim10
$ git log origin/Sim10/2012

# prepare the cherry-pick commit using that commit reference,
$ git cherry-pick -m 1 4e122e6eb38e36652f04fa98dc618c6eaa7d44b8

# merge conflict! see what's wrong
$ git status

# fix the conflict by doing some additional editing
$ vim Conditions/Ecal/Calibration/Gain.xml
$ git add Conditions/Ecal/Calibration/Gain.xml

# finish the cherry-picking (and edit the commit message)
$ git commit -c 4e122e6eb38e36652f04fa98dc618c6eaa7d44b8

$ git push
```

Now go back to the web interface to finish creating the new merge request.

References

- Presentation at OPG 2017-02-23
- Presentation at PPTS 2017-04-10

-- MarcoClemencic - 2017-04-20